M. Tech. (Computer Science) Dissertation Series

Design of an Efficient Content-Based Image Retrieval (CBIR) System

a dissertation submitted in partial fulfillment of the requirements for the M. Tech. (Computer Science) degree of the Indian Statistical Institute

By

Debi Prasad Sahoo [illegible]
28th July 2000.

under the supervision of

Prof. Bhargab B. Bhattacharya
Advanced Computing and Microelectronics Unit

INDIAN STATISTICAL INSTITUTE
203, Barrackpore Trunk Road
Calcutta- 700 035

STATISTICAL INSTITUTE
CALCUTTA

Certificate of Apporval

This is to certify that this dissertation titled **Design of an Efficient Content-Based Imgae Retrieval(CBIR) System** by Debi Prasad Sahoo towards partial fulfillment of the requirement for the M.Tech. programme in Computer Science at the Indian Statistical Institute,Calcutta. This report embodies the work carried out under my supervision. His work is satisfactory.

B. B. Bhattacharya

Prof Bhargab B. Bhattacharya 27/7/00

Advanced Computing and Microelectronics Unit

Indian Statistical Institute,Calcutta.

Acknowledgement

My sincerest gratitude goes to *Prof. Bhargab B. Bhattacharya* for his guidance, advice and especially for suggesting a topic that I found challenging throughout the phase of design, analysis and implementation.

I am thankful to our project assistant *Arijit Bishnu* who always encouraged me to find an accurate retrieval scheme. He gave me a clear-cut idea of the problem to be solved.

Last but not the least I would like to thank *T. V. Sriram*, *Sanjeev Kumar Mishra, Chinmoy Mukhopadhay* and *Kishori Mohan Kanwar* for contributing numerous suggestions throughout the dissertation work.

ABSTRACT

This thesis reports a study on an accurate and fast Content-Based Image Retrieval (CBIR) system which is widely used for representation, storage and retrieval of images. In the proposed method, effort has been made to select discriminating features for image representation. Relevant data structures have been proposed for image database so that efficient retrieval can be effected. Performance evaluation has been carried out over a number of binary images in terms of space and time complexities. Finally comparison has been made with some of the existing CBIR systems.

INTRODUCTION

Content Based Image Retrieval (**CBIR**)[1] is the retrieval of images, to the best approximation, similar to the query image submitted by the user.

In large image databases it is highly inefficient to store all the features that are found in an image. Most image databases are *similarity based*. These systems represent images by using certain image feature spaces, define similarity metrics on thóse feature spaces and do retrieve images similar to the query image. Images are represented as points in the multidimensional feature space. Distance measure, usually Euclidean , is followed as metric for determining the similarity.

Basically two search schemes namely

1. nearest neighbor search
2. range search

are most common in use. The nearest neighbor search gives images that are more similar to the query image. But range search gives images within a region of the feature space as per the feature range specified by the user.

Additional features of traditional CBIR

1. **Nature of data structure**

   A *dynamic indexing structure* supports dynamic updates like insertion, retrieval with each new query whereas a *static structure* works only on the data set given beforehand. Nothing can be done, in the later case, if the query data is not present in the given data set.

2. **Storage Type**

   A *memory resident* structure creates indices in the computer memory, while *disk resident* indexing structure stores indices on hard disk.

3. **Dimension of feature space**

   This defines the number of decision paths taken for retrieving images.

Category of the CBIR proposed

1. Similarity based
2. Euclidean-distance measure for classifying images
3. Search scheme: nearest neighbors search, exact match as well
4. Dynamic data structure: insertion, retrieval and update in the permanent storage device
5. Memory resident data structure
6. One-dimensional search for tested database and multi-dimensional search for modified version of the database.

Desired properties of the image atributes

1. Attributes should be invariant to scale change.
2. Attributes should be invariant to different orientation of the image namely,
   a. Translation
   b. Rotation
3. Attributes are selected in such a manner that their range is wide and spaced more or less uniformly through out the range.

Keeping these 3 points in mind we have selected the following two parameters for our CBIR system.

1. Inverse of compaction factor(area/perimeter square)
2. Number of vertices of the convex hull of the image

Assumptions

1. white portion of the image represents the object (white pixel be 0-bit)
2. black portion of the image represents the background (black pixel be 1-bit)

Inverse of compaction factor

Definition

It is the inverse of the ratio of the area of the image to the square of the perimeter of the image.

Image [i][j]: 2-dimensional matrix for storing the pixel values of the image

Image_edge[i][j]:2-dimensional matrix for storing the boundary pixels of the image

Area= no. of 0's in Image[i][j]

Perimeter= no. of 0's in Image_edge[i][j]

Algorithm

```
        Find_compactionfactor(image[][], height, witdh){
            Image[][], Image_edge[][]: n*n matrix
            Area=0, perimeter=0, temp=0;
                For i 1 to n
                    For j 1 to n
                        If ( image [i][j]=0)
                            Area ++;
/*Area stores the area of the image and perimeter stores the perimeter of the image */
                        If (! outermost pixels){
                            If (eight neighbor pixels are all 0){
                                Temp ++;
                                Image_edge[i][j]=   1;
/* if all the 8-neighbours are object pixel, pixel considered is not an edge pixel */
                            }
                            else
    Image_edge[i][j]=0 ;
/* if one of  the 8-neighbours is not an object pixel, pixel considered is not an edge pixel */
}
else{
    if(image[i][j]=0)
        Image_edge[i][j]=0;
/* outermost pixels whose intensity values = 0 are part of the edge of the image */
```

```
                else
                        Image_edge[i][j]=1;
/* outermost pixels whose intensity values = 1 are part of the edge of the image */
                        }
        Perimeter= area –temp;
        Inv_compaction=(perimeter)^2/area;
        Return(inv_compaction);
}
```

Time Complexity: $O(n^2)$

Number of convex hull vertices [2]

Definition

Convex hull of a set Q of points is the smallest convex polygon p for which each point in Q is either on the boundary of p or in it's interior.

Algorithm

```
                **Chull_points**(*image_edge[][], perimeter*){
                Counter=0;
/* counter stores the number of vertices of the convex hull corresponding to
image_edge[][] */
                Q contains the boundary points as stored in image_edge[][]
                1.p0 be the point in Q with the minimum y-ço-ordinate or the left most
such point incase of a  tie.
                2.<p1,p2,............,pn> be the remaining points in Q sorted by polar angle in
counter clockwise order around p0.(if more than one point has the same angle ,remove all
but the one that is farthest from p0)
                3.top[s]=0
                4.push(p0,s)
                5.push(p1,s)
                6.push(p2,s)
                7.for i=3 to n (perimeter)
```

```
        8. do while the angle formed by points Next_to_top(s),Top(s) and $p_i$ makes a non-left turn
        9. do pop(s)
        10. push(s,$p_i$).
        11. counter ++
        12. return(counter)
    }
```

Time Complexity

Running time = O(nlogn) where n=|Q|

Sorting time in step 2 = O(nlogn) using *heapsort*

Step3 to step6 : O(1) time

Step 8 : O(n)

Since there are atmost m-2 pop operation to be performed and each call of pop takes O(1) time and m<= n-1

Step-10 : O(1) time

For loop in step 7 takes O(n) time exclusive of while loop of line 8-9

DATA STRUCTURE FOR EXACT MATCH

Table1, Table2 : two hash tables created to store the inv_compaction and no_of_convex_hull vertices respectively whenever a new image is inserted into the database.

Collision resolution scheme: chaining

Hash function selected: prime number but not near any power of 2(701)

H(k) = k mod 701

Where k= inv_compaction for table1 and no_of_convex_hull vertices for table2

Input to the CBIR system : image (filename)

Preprocessing steps

1. create table1 and table2

2. read image features stored in the image store (basically from secondary storage device) called imagestore.dat say 1 record.

3. for uniform distribution of data into the hash table the following operation are taken

a. if (attribute 1< 0 or <100) key1 = attribute1*10
b. if (attribute 2< 0 or <100) key2 = attribute2*10

key1 and key2 are used to determine the position of the attributes in the corresponding hash table.

4.Insert (*table1* , *key1* , *attribute1*)
Insert (*table2, key2, attribute2*)

Step 2, 3 and 4 are repeated for all the records of *imagestore.dat* file

Query Session

The user has to give the choice whether he wants to insert the image or search the image in the database. Processing starts only after the submission of the query image by the user.

INSERT

Before inserting the image submitted by the user search is made for the query image.If the *search function* returns 1 reject insert operation saying *exact match found* else *insert function* is invoked .Instead of step2 of preprocessing steps described earlier attribute1 and attribute2 are calculated using **find_compaction ( )** and **chull_points ( )** respectively and then step 3 and step4 are followed.

```
SEARCH:1. attribute1=find_compaction(image, height, width)
          attribute2= chull_points( Image_edge, perimeter)
       2. determine key1 and key2 as described incase of preprocessing steps
       3. invoke search function
              flag1=search(table1, key1, attribute1)
              if(flag=1)/*data match in first table*/{
                     flag2=search(table2, key2, attribute2)
                     if(flag2=1)
                            match found in table1 as well as in table2
                     else
```

```
                    mismatch in attribute2
        }
        else
                infer query image not found due to mismatch in attribute1
```

FUNCTIONS

1. void insertion(*table , key, attribute*){

```
    i. create a temporary list capable of storing attribute and a link to a variable of its
type i.e list.
ii. determine the hashed slot by  k=key%701/* position of the key in the table is
determined */
iii. if table[k]->entry=null /* if key is hashed to an empty position attribute is stored there
*/
        table[k]->entry= attribute;
        table[k]->next=null;
else /* if hashed position is not empty all the entries of the list starting at that position is
moved one postion right */
        temp->entry= table[k]->entry;
        temp->next=table[k]->next;
        table[k]->next=temp;
}
```

2. int search (*table, key, attribute*){

```
    i.      k = key % 701;/* position of the key in the hash table is determined */
    ii.     temp = table[k];
    iii.    if table[k]->num=0 /* if the hashed position entry is null */
                    return (0);/*attribute not present in the database*/
            else { /* traverse the list starting with the hashed position and entry at each
    position is compared with the attribute. This is done till the end of that list */
                    do {
                            if(temp->num != attribute){
                                    if(temp->next == null){
                                            flag=0;
```

```
                        break;
                    }
                    else{
                        temp=temp->next;
                        flag=0;/*so far attribute has not been found

                    }
                }
                else{
                    flag=1;/* exact match found*/
                    break;/* matching process is halt at this point */
                }
            } while ( temp ! = null);
    return(flag);}/*end of search function */
```

Complexity Analysis (*average case*)

Let hash table concerned has <u>m slots</u> and it stores <u>n elements</u>

Load factor(α): it is the average number of elements stored in a chain

$$\alpha = n/m$$

Assumption: Hash value h(k) can be computed in O(1) time

With the above assumption we can say time required to search for an element with key k depends linearly on the length of the list T[h(k)]

Two cases arise namely,

1. unsuccessful: no element in the table has key k
2. successful: finds an element with key k

Unsuccessful: under uniform simple hashing , any key k is equally likey to hash any of the m slots. The average time to search successfully for a key is thus the average time to search to the end of the m lists which is nothing but load factor α=<u>n/m</u>

Number of elements examined = α

Total time required = O ($1+\alpha$) including time for computing h(k)

Successful: Expected number of elements examined is given by the average over the n times in the table, of 1 plus the expected length of the list to which i'th element is added . The expected length of that list is (i-1) /m.

So expected number of elements examined in a successful search is

$1/n(\Sigma (1 + (i-1) /m))$ for $i=1$ to n

$= 1 + (1/(nm))(n*(n-1)/2)$

$= 1 + \alpha/2 - 1/(2m)$

$= O(1+\alpha)$

if $n= O(m)$

$\alpha=n/m$

$=O (m)/m$

$=O (1)$

Thus searching takes constant time on the average.

Insertion

New key is inserted at the beginning of the list. So it takes O (1) running time in the worst case.

IMAGE DATABASE FOR RETRIEVING IMAGES ON THE BASIS OF NEIGHBORHOOD SEARCH

Image plane

Plane created by taking attribute1 and attribute2 as its two reference axes. So each image is mapped to a point on this plane.

Euclidean distance

let p_1, p_2 be two points on the image plane

E- be Euclidean distance between p_1, p_2

$(a1)_{p1}$-attribute1 corresponding to point p1

$(a2)_{p1}$-attribute2 corresponding to point p1

$(a1)_{p2}$-attribute1 corresponding to point p2

$(a2)_{p2}$-attribute2 corresponding to point p2

$E = (((a1)_{p2}-(a1)_{p1})^2 + ((a2)_{p2}-(a2)_{p1})^2)^{1/2}$

We are using a certain minimum distance called *min_class_distance* for classifying images.

Data structure

A link list of class representative is created . The class representative is basically the node created at the time when a new class is formed. The class representative is the root of the binary search tree that holds images of its class while other members of the class are either internal nodes or leaf nodes.

Class_list: It is defined as a structure as

```
        Struct class_list{
                Int meanx, meany, no, load;
                Struct node *class_root;
                Struct class_list *next;
        };
```

Description of the class_list structure

1. meanx, meany represent the mean point of a class called *class_mean*.

This is used for determining the Euclidean distance between the query image and *class_mean* of existing classes.

2. no represents the class number called *class_no*

3. class_root is a pointer to a variable of type *node* which is created when a new class is formed.

4. next is a pointer to the class next to the current class in the list of classes.

5. load gives the number of images present in the class at any point of time.

Node: It is defined as a structure as

```
Struct node{
Int num1,num2;
Char *filename;
Struct node *left,*right;
};
```

num1, num2 are the attributes of an image which are inserted as image content into the node of a tree representing a class.

Filename is a pointer to the image file.

Left, right are the links of type node to form the left and right child of the node concerned.

Image_store: This is a structure used to download the image features stored in the secondary storage device into the image database created.

```
Struct image_store{
        Char filename[];
        Int num1, num2;
};
```

filename corresponds to image that has been inserted into the database earlier

num1, num2 are attributes of the image that has been inserted into the database earlier.

Steps for insertion

```
1. calculate attribute1 and attribute2 of the query image using function find_compactionfactor(image[][], height, width) and Chull_points(image_edge[][], perimeter)
.
2. if ( list of class is empty){
        a. create a new class
           class = create_class( )
        b. invoke function insert( class→root, num1, num2, filename) to create a search tree of
           the new class created in step 2(a)
        c. Class mean is the point corresponding to the image inserted first into the class.
           class→meanx=num1;
           class→meany=num2;
           class→load=1;
        d. this class is the starting of the list of classes
```

```
}
        else{ /* if class list is not empty then the class of the query image is determined. In case
of exact match insertion operation is rejected. A new class is created for the query image if it
does not belong to any of the existing classes. */
            a. n= Retrieve_images(filename, num1, num2)
            b. if(n=1)
                    reject inferring the presence of the image in database
            else
                    insert the image in it's class by invoking the function,
m=Insert_in_it_class(num1, num2, filename)
m gives the the class_number in which the query image inserted
```

Steps for image retrieval

```
1. Calculate attribute1 and attribute2 of the query image as
num1=find_compaction(image, height, width)
num2=chull_points(image_edge, perimeter)
2.if there is no class
        it is inferred that database is empty
else
        invoke function Retrieve_iamges as
        m=Retrieve_images(filename, num1, num2)
if(m=0)
        it is inferred that no image found in the neighborhood of the query image.
```

Details of the functions used

```
(i)insert_ in_it_class(num1, num2, filename){
1.define class_list *new_class,*class
2.class = find_class(num1, num2)
class is the pointer to the list of type class_list into which image with num1 and num2 falls.
3.if(class != null){
        i.      invoke insert(class→root, num1, num2, filename)
                This inserts the query image into the search tree of its class.
        ii.     update_mean(class, num1, num2)/*after inserting a new image class mean is
                updated */
        iii.    return(class→no)/* this returns the class number into which the image is inserted
                */
        }
        else{
         i. create a new class by , class= **create_class( )**
        ii.     **insert**(*new_root, num1, num2, filename*)/*new_root is assigned null initially */
        iii.    class→root=new_root
        iv.     class→meanx=num1
        v.      class→meany=num2
        }
    }
    }
```

```
(II)find_class(num1, num2){
        1. create class_list variable * class,*temp;
        2. traverse the class_list and determine the Euclidean-distance of the
           query image from the class_mean of each class.
           a[i] =Euclid_dist(class→meanx, class→meany, num1, num2)
           b[i]=class

        3. sort the array a[ ]. While sorting swapping is done in between two
           elements of b[ ] with swapping of two elements of a[ ]. A[0] will the
           minimum Euclidean distance .
        4. b[0] is be the class where the query image would be inserted provided
           a[0] is less than minimum_class_distance.
           If(a[0]<minimum_class_distance)
              Return(b[0]) /* it gives the class in which the input image is to be
           inserted */
           else
              Return(null)/*it indicates a new class is required to be formed for
           inserting the query image.
}
(III)insert(node, num1, num2, filename){
                        if(node = null){/*if input image does not belong to any of
        the existing classes a new binary tree is constructed to construct a new
        class */
              (i)     create a new tree with root node new_root
              (ii)    new_root→num1= num1
              (iii)   new_root→num2= num2
              (iv)    new_root→left = null
              (v)     new_root→right= null
              (vi)    new_root→filename=filename
              }
```

```
else{/* tree corresponding to the class returned by the find_class( ) is traversed to find the position for inserting image attributes into that class */
        if ( num1 < node→num1)
            insert(node→left, num1, num2, filename)
        else
            insert(node→left, num1, num2,  filename)
        }
```

(IV)Update_mean(*class, num1, num2*){

/* this function updates the mean of the class to which a new image has been inserted. */

```
        1. define two temporary variaables temp1,temp2
        2.temp1 = class→meanx * claass→load
        3.temp2 = claass→meany * class→load
        4.new class→load = class→load + 1
        5. class→meanx = ( temp1 + num1 )/ class→load
           class→meany = ( temp2 + num2 )/ class→load
        }
```

(V)Retrieve_images(*filename, num1, num2*){

/* for retrieving images similar to the query image its class is determined and then the tree corresponding to that class is traversed */

```
        1.  find the class of the query image by
                class = find_class( num1, num2)
        2.  if ( class ! = null )
          m=inorder_traversal(class→class_root, filename, num1, num2)
           else
                m = 0/* it indicates the query image does not belong to any of the existing classes */
        3.  return( m )
        }
```

```
(VI)inorder_traversal( node, filename, num1, num2){
            1.tag =0 /* tag is used to check exact match */
              if( node ! = null ){
            2.m=inorder_traversal(node→left, filename, num1, num2)
            3.if( (node→num1 = num1) and (node→num2 = num2))
                    it is concluded that exact match is found and tag is set to 1
            4.print filename as lying in the neighborhood of the query image
            5. m=inorder_traversal ( node→right, filename, num1 ,num2) }
return(tag)}
```

MODIFIED VERSION OF THE IMAGE DATABASE *(FOR ACCURATE RETRIEVAL)*

For accurate retrieval we define one variable called *radius_of_neighborhood*.What is done at the time of retrieval is search is made in the circular region formed in the *image plane* with query image as the center and *radius_of_neighborhood* as the radius.

Note

We call the point(in the *image plane*) corresponding to the query image as *query point* and class mean as *mean_point* of that class.

We are considering only those classes those satisfying the *condition*:

Euclidean distance between the *mean_point* of the class and the *query point* is less than the *minimum_distance_of_neighbouring_class* and call them *most_close_neighbours* .

Modification in the algorithmic steps

In find_class ( ) function:

1. a temporary variable called *class_counter* is used to store no_of _class satisfying neighborhood condition defined earlier.
2. In sorted arrary determine classes satisfying neighborhood condition and increament the *class_counter* for each of the class found so.

The above two steps are included in *step-3* of the **find_class** function described earlier.

In Retrieve_image ( ) function:

Step-2 is modified to

```
For I = 0 to class_counter
        M=inorder_traversal(b[I]→class_root, filename, num1 ,num2)
```

In inorder_traversal( ) function:

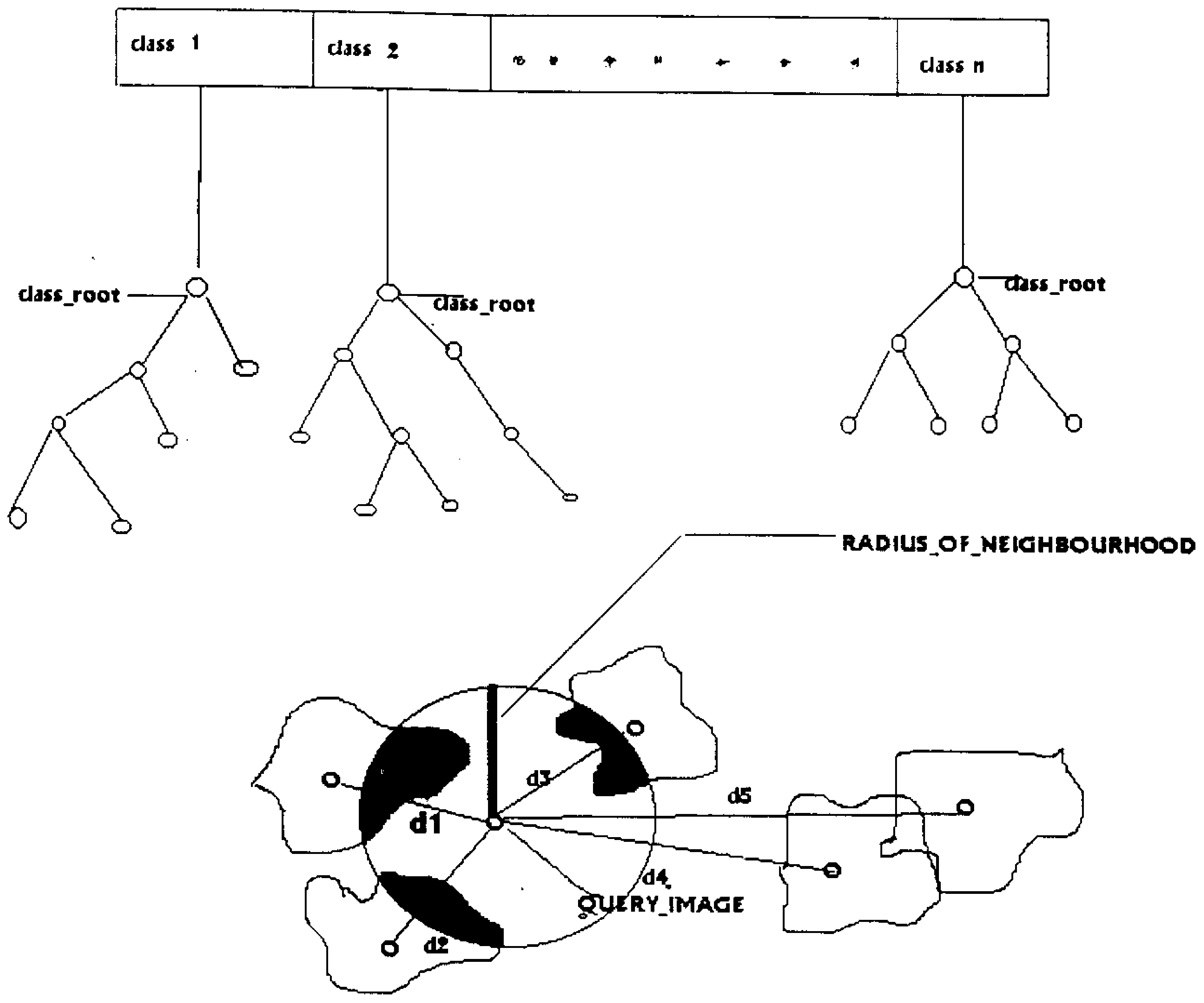
The only variation in this function is in making decision whether an image of the class being traversed falls within the *circular_neighborhood* region of the query iamge or not.

So *step-3* is modified to

If **Euclidean_distance**(*node*$\rightarrow$*num1*,*node*$\rightarrow$*num2*,*num1*,*num2*) <*radius_of_neighbourhood* accept the image present in the node as similar to the query image.

For more accurate result of retrieval *radius_of_neighbourhood* is reduced with each iteration of the retrieval of the same query image during *query_session* until **find_class**( ) returns null.

PICTORIAL REPRESENTATION OF RETRIEVAL SCHEME:



d1,d2 and d3 are less than minimum_distance_of_neighouring_class and hence class 1 ,class2 and class 3 considered for retrieval and class corresponding to minimum among d1,d2 and d3 is considered for insertion of the query image

TEST RESULT

We have got the following result after testing the *proposed CBIR* (*neighbourhood search method*) using standard binary images.

Note

Time measured excluding the time taken for image feature computation.

Total number of binary image stored in the image database=*384*

Average number of clock cycles required for inserting an image into the image database=*1718*

Average number of clock cycles required for retrieving images similar to the query image=*1484*

CLOCKS_PER_SEC=10^6

Average time for inserting an image into the image database =*$1.718*10^{-3}$ seconds*

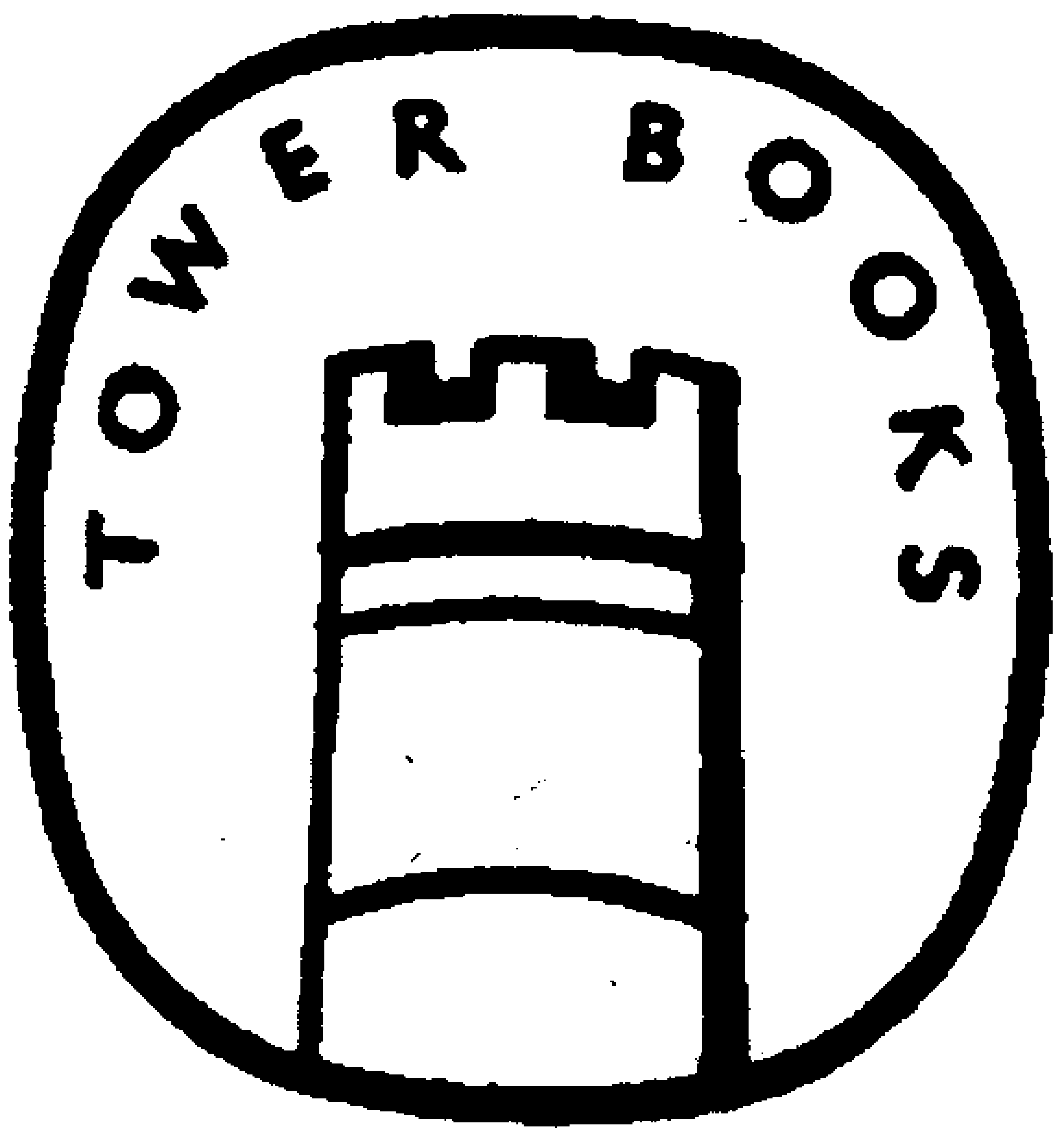
Average time for retrieving images most close to the query image=$1.484*10^{-3}$ seconds

Average search space

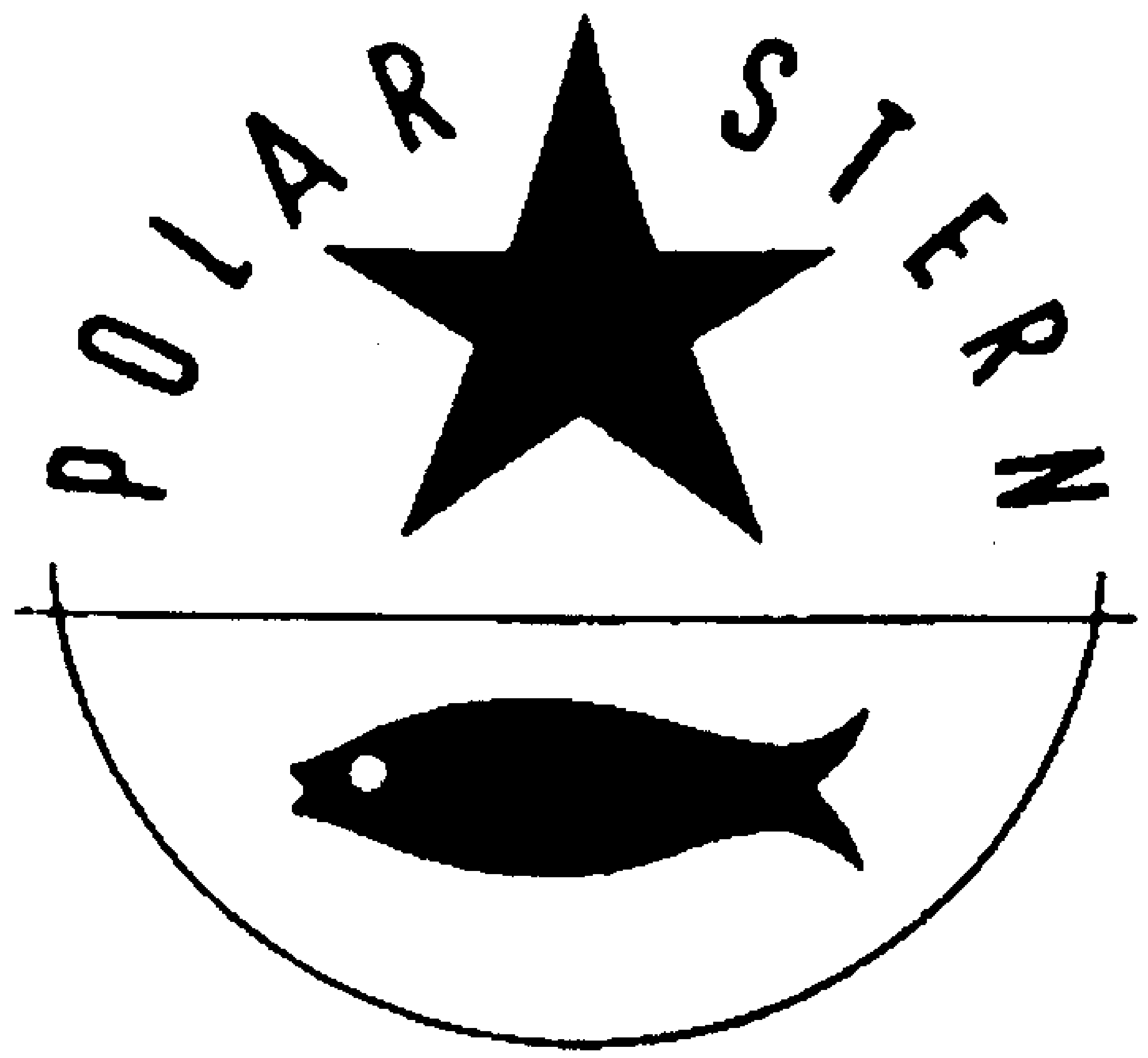
It is defined as the average number of images examined while inserting the query image into the image database as well as retrieving images most similar to the query image.

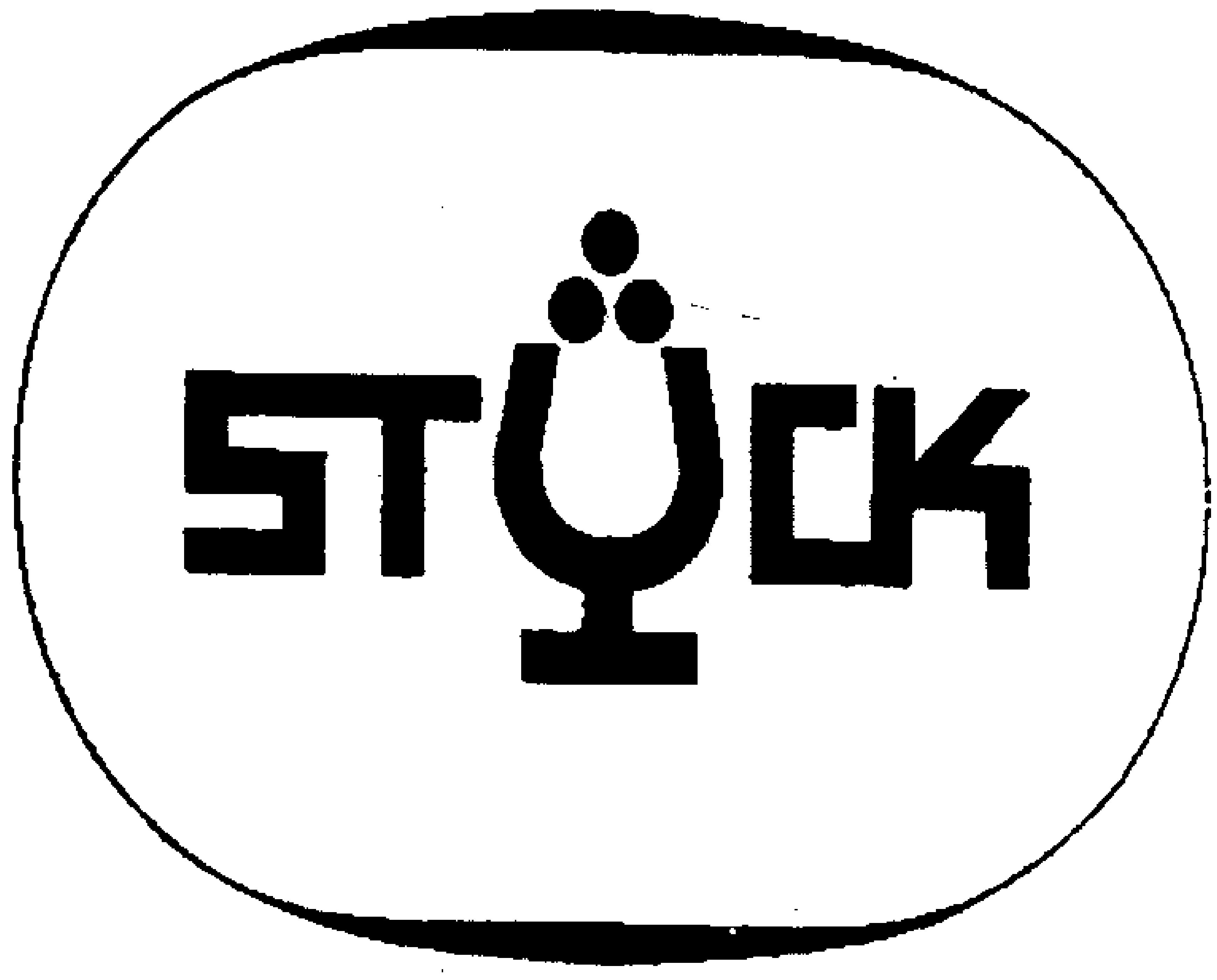
For the system tested,

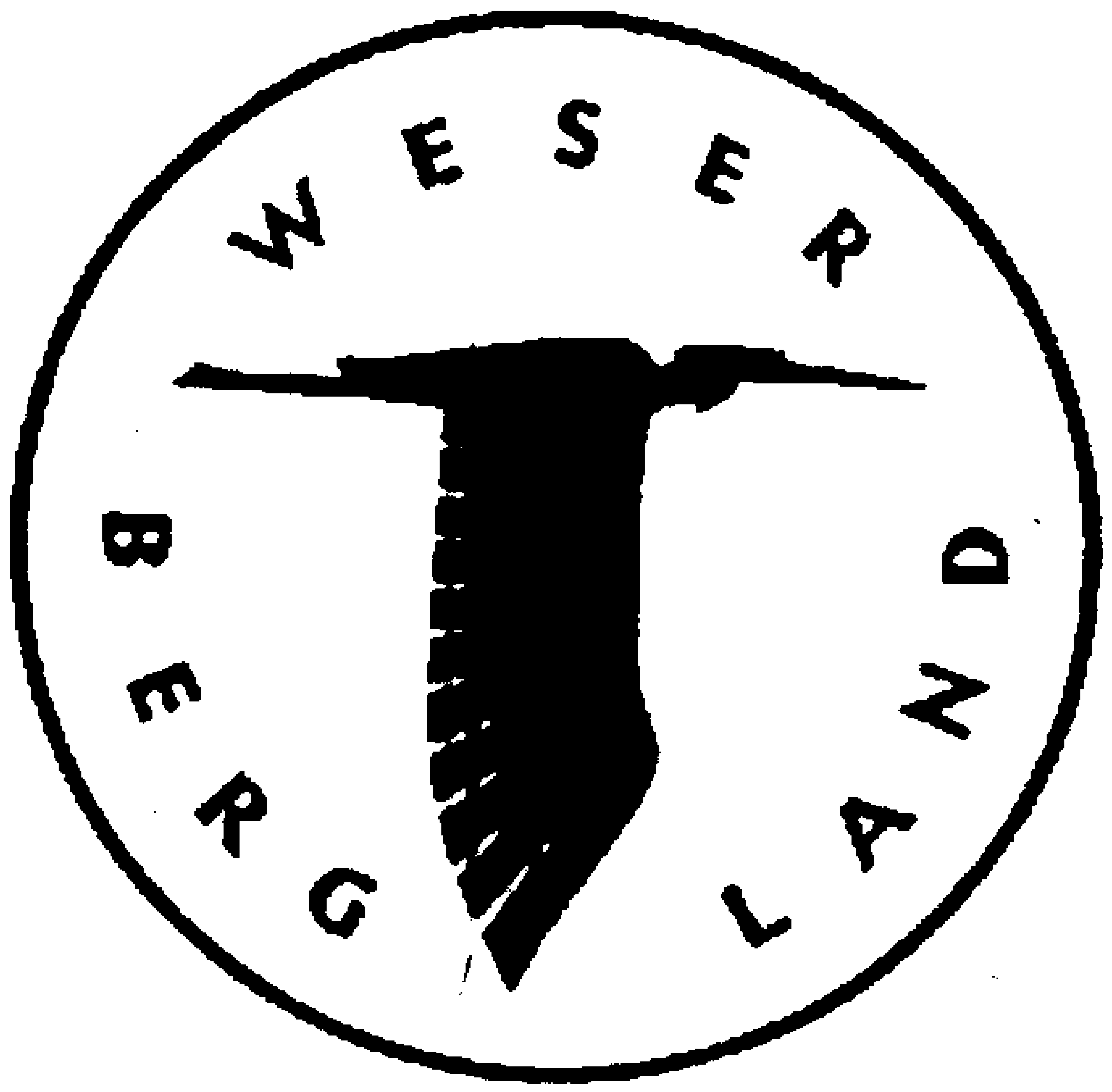
Average search space for insertion ≅average search space for retrieval=20

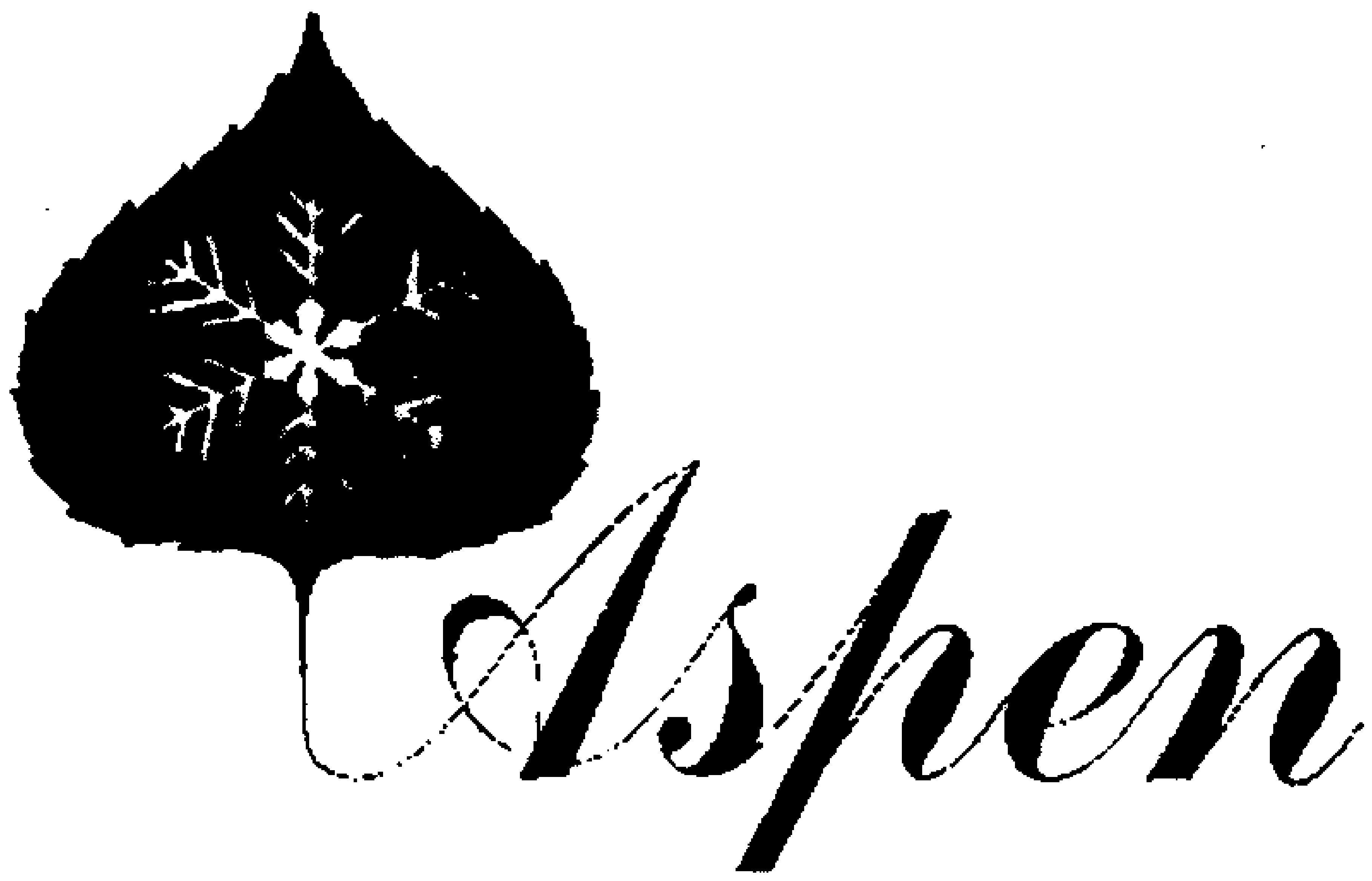


QUERY-IMAGE



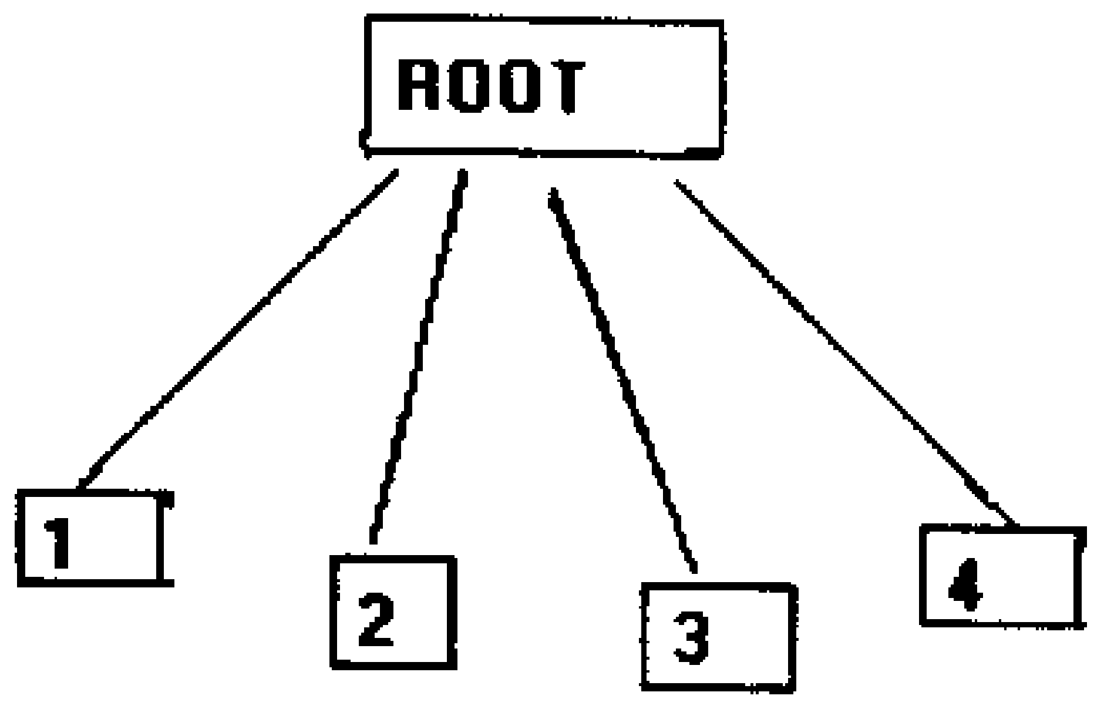


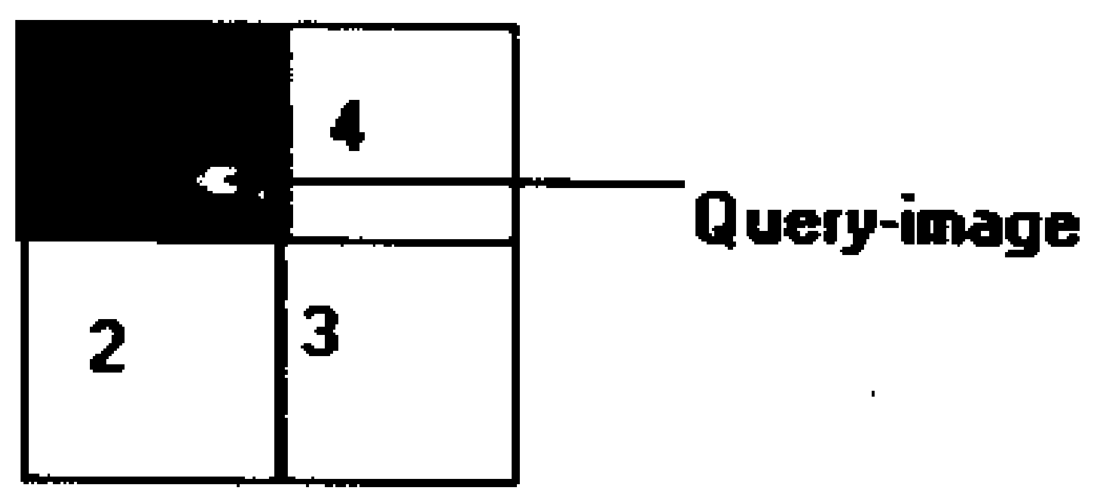




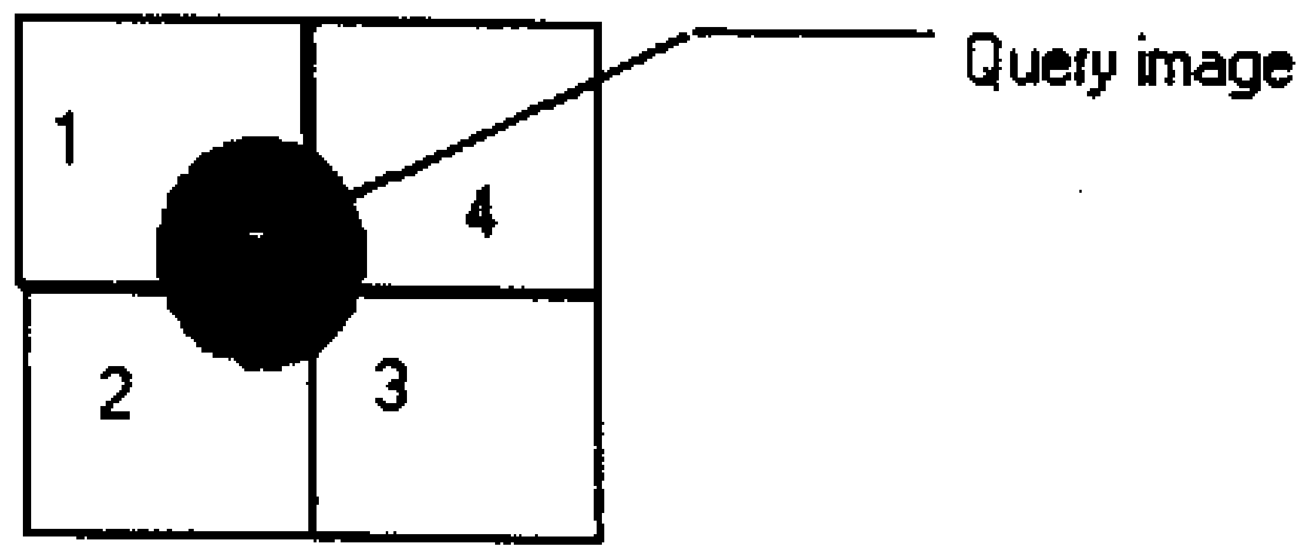
Comparisions of standard indexing structure with the retrieval scheme proposed:

KDB-TREE:





If the query image falls in the region of class1 as shown in the figure KDB-tree retrieves image of class1 only since there is no overlap between any pair of regions and each data has only one path beween itself and the root.Unlike KDB-tree our retrieval scheme will retrieve image falling in the shaded region of the figure shown below.(number of image retrieved depends on the radius_of_neighbourhood)



R-TREE

In R-tree the search algorithm descends the tree from the root to the find all the data objects whose enclosing rectangles overlap a search rectangle.

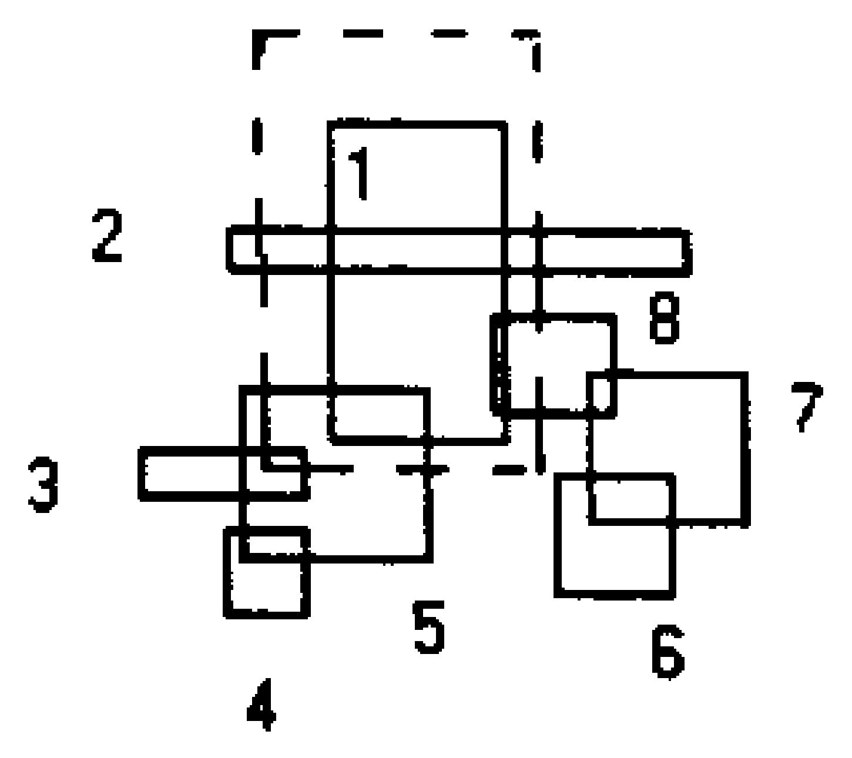
Let t be the root of an R-tree,s be search rectangle specified by the user. The search is performed by the following recursive operation:

1. if t is not a leaf, find all the entries e whose rectangle e.rect overlaps s . For each overlapping entry set t=e.np and same operation is perform recursively.

2. If t is a leaf , find all the entries e whose rectangle e.rect overlaps s, and retrieve the record that is pointed to by e.id from the database.

Demerit: more burden on the user since he has to select the search rectangle and bounding rectang les at the leaf nodes are not optimised.

Example: If the dotted rectangle is the search rectangle then images present in the rectangular regions overlapping with the search rectangle is retrieved.Here images corresponding to class 1, 2 ,3 , 5 and 8 are retrieved.



R*-TREE

It share the same tree structure with R-tree,but achieves a better performance by introducing more sophisticated optimization criteria for node splitting.

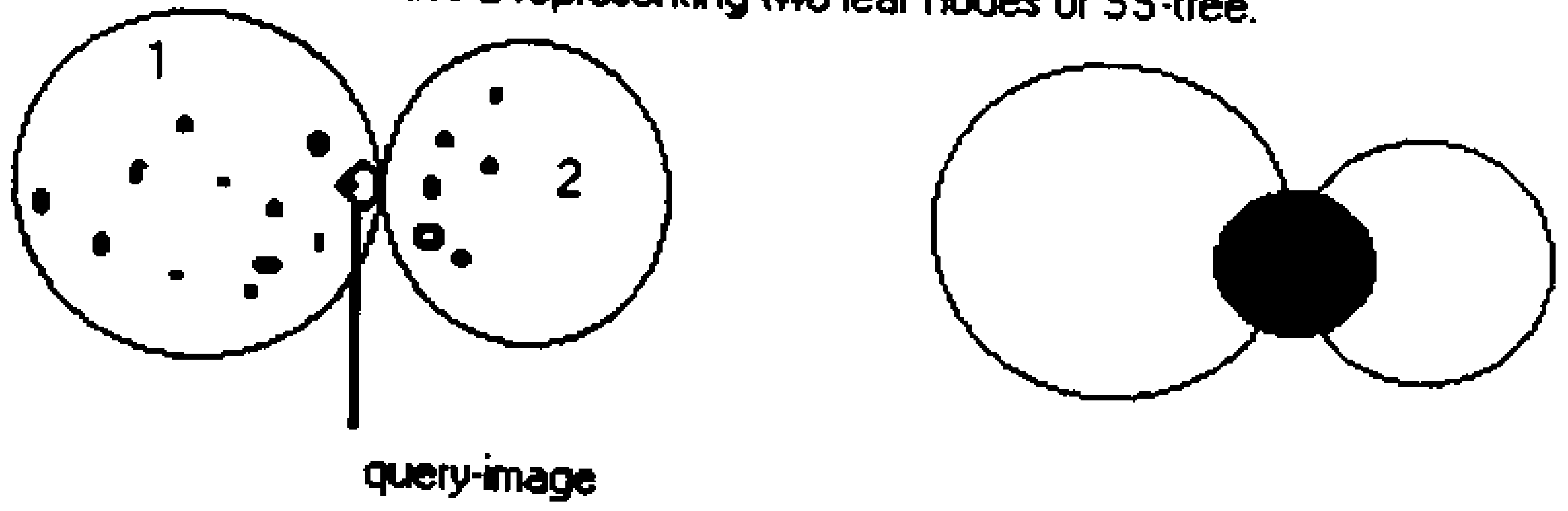
Demerit

1.Extra overhead in terms of node splitting and forced re-insertion

SS-TREE

Tree makes use of minimum bounding spheres instead of minimum bounding rectangles as in case of R-tree and R*-tree.

Example· consider two classes class-1 and class-2 representing two leaf nodes of SS-tree.



Corresponding to the query image shown SS-tree retrieves images of class-1 only where as our retrievel scheme retrieves image of both classes depending on the radius_of_neighbouhood (shown as the shaded region).

APPLICATION

Secured Network (model proposed)

The concept of *"seeing is believing"* can readily be applied to the computer system when it comes to recognizing people. Content-based image retrieval (*CBIR)* can play crucial role in building a system that will allow only permitted users to access any device connected to the computer network.

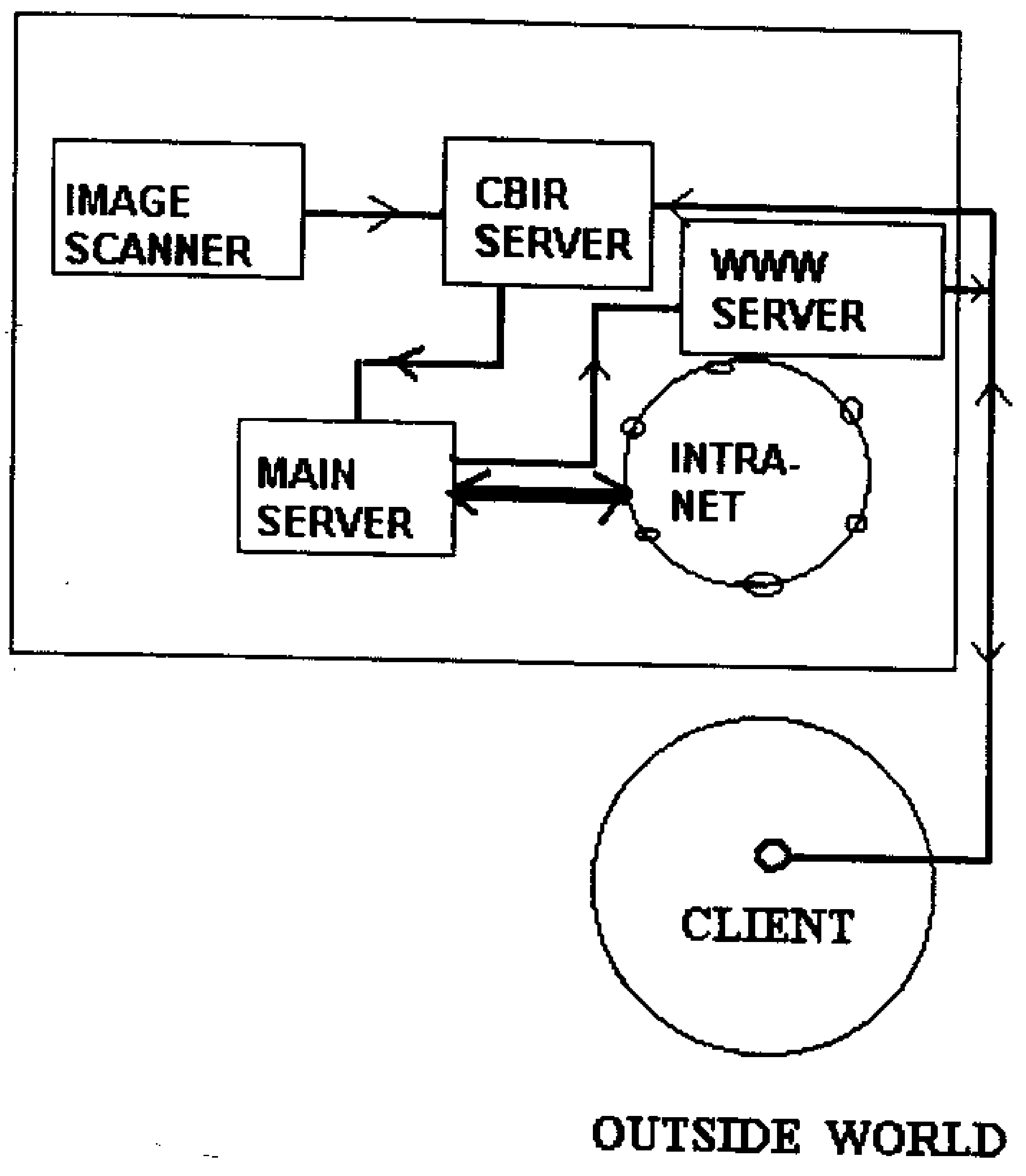
Steps involved

1. Image database is constructed to store discriminating features of permitted users.
2. Features for existing users are computed and stored in the image database.
3. The user, who wants to access to any computer system connected to the network, has to submit his picture to a scanner.
4. The image scanned is fed as input to the server running the *CBIR Software.*
5. If CBIR *Server* finds a match in the image database then status of the terminal requested for is checked by the *server* .If it is found that the computer is readily available and the user has the permission to access it then access is given to that user.
6. *CBIR Server* waits for the next request.

Note

1. If any user logged into the Intranet is interested in accessing the outside world request is passed to the *WWW Server.*
2. Step 1 to step 6 is applicable to all users from outside world for accessing the *secured network.*

MODEL OF A SECURED NETWORK SYSTEM



CONCLUSION

The *Content-Based Image Retrieval* system proposed is efficient in terms of *average insertion time*, *average retrieval time* and *search space*. The system has been implemented *in SUN Workstation*. Interface has been made for demonstration of image retrieval. Only thing that need improvement is better user interface for giving the query image as input to the system proposed. *CBIR*[1] systems will play an important role in applications such as *network security*, *private information retrieval* and *forensic science*.

References

[1] Yihong Gong. Intelligent Image Databases Towards Advanced Image Retrieval.

[2] T. H. Cormen, Charles E. Leiserson and Ronald L. Rivest. Introduction to Algorithms.

[3] Rafael C. Gonzalez and Richard E. Woods. Digital Image Processing.